Unit 4 : Spark

Q.1 What is Spark? Explain the features of Spark.

❖ Spark

- Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

- Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

❖ Features of Spark

- **Lighting Fast Processing**
  - » Spark enables applications in Hadoop clusters to run up to 100x faster in memory, and 10x faster even when running on disk. Spark makes it possible by reducing number of read/write to disc.
  - » It stores this intermediate processing data in-memory. It uses the concept of an Resilient Distributed Dataset (RDD), which allows it to transparently store data on memory and persist it to disc only it's needed. This helps to reduce most of the disc read and write – the main time consuming factors – of data processing.
- **Ease of Use as it supports multiple languages**

Spark lets you quickly write applications in Java, Scala, or Python. This helps developers to create and run their applications on their familiar programming languages. It comes with a built-in set of over 80 high-level operators. We can use it interactively to query data within the shell too.

- **Support for Sophisticated Analytics**

  In addition to simple "map" and "reduce" operations, Spark supports SQL queries, streaming data, and complex analytics such as machine learning and graph algorithms out-of-the-box. Not only that, users can combine all these capabilities seamlessly in a single workflow.

- **Real Time Stream Processing**
  - » Spark can handle real time streaming. Map-reduce mainly handles and process the data stored already. However Spark can also manipulate data in real time using Spark Streaming. Not ignoring that there are other frameworks with their integration we can handle streaming in Hadoop.
  - » Easy: Built on Spark's lightweight yet powerful APIs, Spark Streaming lets you rapidly develop streaming applications
  - » Fault tolerant: Unlike other streaming solutions (e.g. Storm), Spark Streaming recovers lost work and delivers exactly-once semantics out of the box with no extra code or configuration
  - » Integrated: Reuse the same code for batch and stream processing, even joining streaming data to historical data
- **Ability to Integrate with Hadoop and Existing HadoopData**
  - » Spark can run independently. Apart from that it can run on Hadoop 2's YARN cluster manager, and can read any existing Hadoop data. That's a BIG advantage! It can read from any Hadoop data sources for example HBase, HDFS etc.

» This feature of Spark makes it suitable for migration of existing pure Hadoop applications, if that application use-case is really suiting Spark. As Spark is using immutability more all scenarios might not be suitable for migration.

- **Active and Expanding Community**

  Apache Spark is built by a wide set of developers from over 50 companies. The project started in 2009 and as of now more than 250 developers have contributed to Spark already! It has active mailing lists and JIRA for issue tracking.

Q.2 Explain Component of Spark.

❖ Spark Core Component

- Spark Core component is the foundation for parallel and distributed processing of large datasets.
- Spark Core component is accountable for all the basic I/O functionalities, scheduling and monitoring the jobs on spark clusters, task dispatching, networking with different storage systems, fault recovery and efficient memory management.
- Spark Core makes use of a special data structure known as RDD (Resilient Distributed Datasets).
- Data sharing or reuse in distributed computing systems like Hadoop MapReduce requires the data to be stored in intermediate stores like Amazon S3 or HDFS. This slows down the overall computation speed because of several replications, IO operations and serializations in storing the data in these intermediate stable data stores.

- Resilient Distributed Datasets overcome this drawback of Hadoop MapReduce by allowing - fault tolerant 'in-memory' computations.

❖ Spark SQL Component

- Spark developers can leverage the power of declarative queries and optimized storage by running SQL like queries on Spark data, that is present in RDDs and other external sources.
- Users can perform, extract, transform and load functions on the data coming from various formats like JSON or Parquet or Hive and then run ad-hoc queries using Spark SQL.
- Spark SQL eases the process of extracting and merging various datasets so that the datasets are ready to use for machine learning.
- DataFrame constitutes the main abstraction for Spark SQL. Distributed collection of data ordered into named columns is known as a DataFrame in Spark. In the earlier versions of Spark SQL, DataFrame's were referred to as SchemaRDDs.

❖ Spark Streaming

- Spark Streaming is a light weight API that allows developers to perform batch processing and streaming of data with ease, in the same application.
- Discretized Streams form the base abstraction in Spark Streaming. It makes use of a continuous stream of input data (Discretized Stream or Stream- a series of RDD's) to process data in real-time.
- Spark Streaming leverages the fast scheduling capacity of Apache Spark Core to perform streaming analytics by ingesting data in mini-batches. Transformations are applied on those mini batches of data.

- Data in Spark Streaming is ingested from various data sources and live streams like Twitter, Apache Kafka, Akka Actors, IoT Sensors, Amazon Kinesis, Apache Flume, etc. in event drive, fault-tolerant and type-safe applications.

❖ Spark Component MLlib

- MLlib is a low-level machine learning library that can be called from Scala, Python and Java programming languages.
- MLlib is simple to use, scalable, compatible with various programming languages and can be easily integrated with other tools.
- MLlib eases the deployment and development of scalable machine learning pipelines.
- MLlib library has implementations for various common machine learning algorithms –
  - » Clustering- K-means
  - » Classification – naïve Bayes, logistic regression, SVM
  - » Decomposition- Principal Component Analysis (PCA) and Singular Value Decomposition (SVD)
  - » Regression –Linear Regression
  - » Collaborative Filtering-Alternating Least Squares for Recommendations

❖ Spark GraphX

- GraphX is an API on top of Apache Spark for cross-world manipulations that solves this problem.
- Spark GraphX introduces Resilient Distributed Graph (RDG- an abstraction of Spark RDD's). RDG's associate records with the vertices and edges in a graph. RDG's help data scientists perform several graph

operations through various expressive computational primitives. These primitives help developers implement Pregel and PageRank abstraction in approximately 20 lines of code or even less than that.

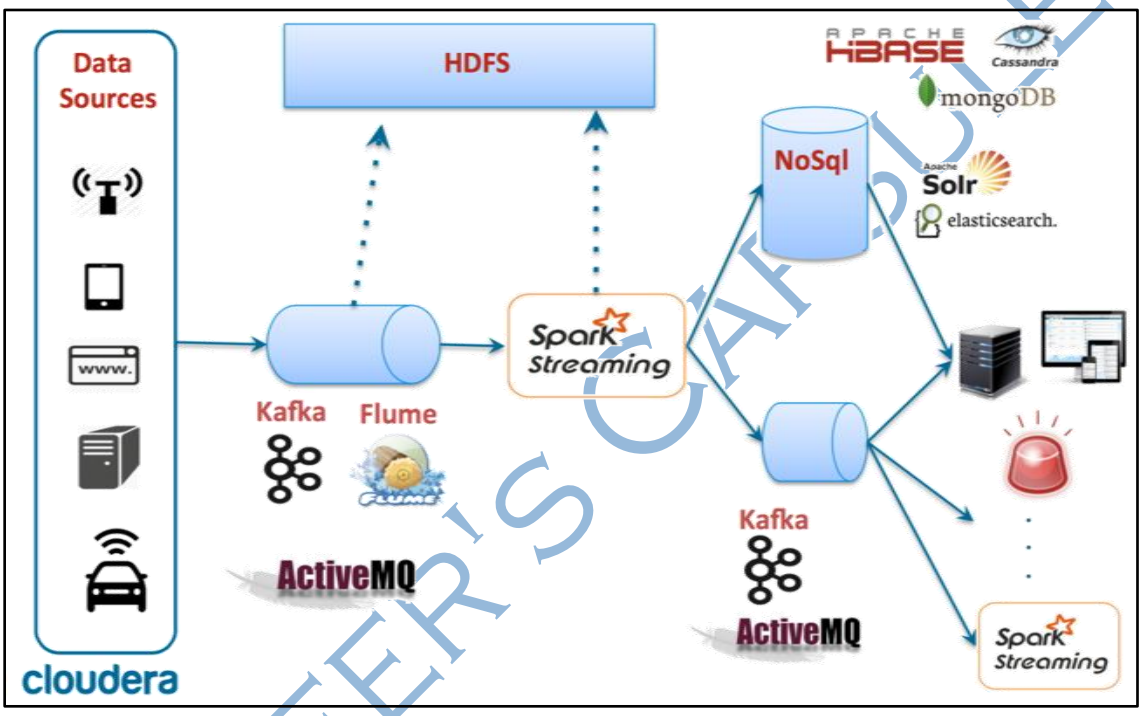
- GraphX component of Spark supports multiple use cases like social network analysis, recommendation and fraud detection.
- Other graph databases can also be used but they require several systems to create the entire computation pipeline, Using Spark GraphX, data scientist can work with graph and non-graph sources to achieve flexibility and resilience in graph computing.

Q.3 Explain stream processing architectures.

A stream processing architecture is typically made of a following components.

- **Data sources** – source of data streams. This is the raw data and so the source of truth. A data source could be a sensor network, or a mobile application, or a web client, or a log from a server, or even a thing from Internet of Things.
- **Message bus** – reliable, high-throughput and low latency messaging system. Kafka and Flume are obvious choices. There are many other options as well, like, ActiveMQ, RabbitMQ, etc. Kafka is definitely gaining lots of popularity right now.
- **Stream processing system** – a computation framework capable of doing computations on data streams. There a few stream processing frameworks out there, Spark Streaming, Storm, Samza, Flink, etc. Spark Streaming is probably the most popular framework for stream processing right now.

- **NoSql store** – processed data is of no use if it does not serve end applications or users. End applications like to do lots of fast read and writes. HBase, Cassandra, MongoDb are popular choices.
- **End applications** – these are the application which consume the processed data/result streams.



Q.4 What is RDD? Explain properties of RDD.

- **RDD**
  - Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects.
  - Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.
  - RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

- RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

❖ **Properties of RDD**

- **Immutable (Read only cant change or modify)** - Data is safe to share across processes. It can be created or retrieved anytime which makes caching, sharing & replication easy. It is a way to reach consistency in computations.
- **Partitioned** - It is basic unit of parallelism in RDD. Each partition is logical division of data/records.
- **Coarse gained operations** - it's applied to any or all components in datasets through maps or filter or group by operation.
- **Action/Transformations** - All computations in RDDs are actions or transformations.
- **Fault Tolerant** - As the name says or include Resilient which means its capability to reconcile, recover or get back all the data (coarse/fine grained & low overhead) using lineage graph.
- **Cacheable** - It holds data in persistent storage (memory/disk) so that they can be retrieved more quickly on the next request for them.
- **Persistence** - Option of choosing which storage will be used either in-memory or on-disk.
- **Lazy evaluated** - i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Parallel** - i.e. process data in parallel.

- **Typed** - RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].
- **Location-Stickiness** - RDD can define placement preferences to compute partitions (as close to the records as possible).
- **Distributed** - with data residing on multiple nodes in a cluster.

Q.5 What are the ways of creating RDD?

- There are three ways to create an RDD in Spark.
  - » Parallelizing already existing collection in driver program.
  - » Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
  - » Creating RDD from already existing RDDs.

1) Parallelized collection (parallelizing)

- In the initial stage when we learn Spark, RDDs are generally created by parallelized collection.
- By taking an existing collection in the program and passing it to SparkContext's parallelize() method. This method is used in the initial stage of learning Spark since it quickly creates our own RDDs in Spark shell and performs operations on them.
- This method is rarely used outside testing and prototyping because this method requires entire dataset on one machine.
- Consider the following example of sortByKey(). In this, the data to be sorted is taken through parallelized collection:

```
val rdd1 = spark.sparkContext.parallelize(Array("jan","feb","mar","april","may","jun"),3)

val result = rdd1.coalesce(2)
```

```
result.foreach(println)
```

2) External Datasets (Referencing a dataset)

- In Spark, distributed dataset can be formed from any data source supported by Hadoop, including the local file system, HDFS, Cassandra, HBase etc. In this, the data is loaded from the external dataset.
- To create text file RDD, we can use SparkContext's textFile method. It takes URL of the file and read it as a collection of line. URL can be a local path on the machine or a hdfs://, s3n://, etc.
- The point to noet down is that the path of the local file system and worker node should be same. The file should be present at same destinations both in local file system and worker node. We can copy the file to the worker nodes or use a network mounted shared file system.
- DataFrameReader Interface is used to load a Dataset from external storage systems (e.g. file systems, key-value stores, etc). Use SparkSession.read to access an instance of DataFrameReader.DataFrameReader supports many file formats-

a) csv (String path)

It loads a CSV file and returns the result as a Dataset<Row>.

Example:

```
import org.apache.spark.sql.SparkSession
def main(args: Array[String]):Unit = {
```

```
object DataFormat {
val spark =
SparkSession.builder.appName("AvgAnsTime").master("local").getOrCreate()
val dataRDD = spark.read.csv("path/of/csv/file").rdd
```

b) json (String path)

It loads a JSON file (one object per line) and returns the result as a Dataset<Row>

```
val dataRDD = spark.read.json("path/of/json/file").rdd
```

c) textFile (String path)

It loads text files and returns a Dataset of String.

```
val dataRDD = spark.read.textFile("path/of/text/file").rdd
```

3) Creating RDD from existing RDD

- Transformation mutates one RDD into another RDD, thus transformation is the way to create an RDD from already existing RDD. This creates difference between Apache Spark and Hadoop MapReduce.
- Transformation acts as a function that intakes an RDD and produces one. The input RDD does not get changed, because RDDs are immutable in nature but it produces one or more RDD by applying operations. Some of the operations applied on RDD are: filter, count, distinct, Map, FlatMap etc.

Example:

```
val
words=spark.sparkContext.parallelize(Seq("the", "quick", "brown", "fox", "jumps", "
over", "the", "lazy", "dog"))

val wordPair = words.map(w => (w.charAt(0), w))

wordPair.foreach(println)
```

Q.6 How spark is faster than Mapreduce?

- One of the main limitations of MapReduce is that it persists the full dataset to HDFS after running each job. This is very expensive, because it incurs both three times (for replication) the size of the dataset in disk I/O and a similar amount of network I/O.
- Spark takes a more holistic view of a pipeline of operations. When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage. This is an innovation over MapReduce that came from Microsoft's Dryad paper, and is not original to Spark.
- The main innovation of Spark was to introduce an in-memory caching abstraction. This makes Spark ideal for workloads where multiple operations access the same input data. Users can instruct Spark to cache input data sets in memory, so they don't need to be read from disk for each operation.
- What about Spark jobs that would boil down to a single MapReduce job? In many cases also these run faster on Spark than on MapReduce. The primary advantage Spark has here is that it can launch tasks much faster.

- MapReduce starts a new JVM for each task, which can take seconds with loading JARs, JITing, parsing configuration XML, etc. Spark keeps an executor JVM running on each node, so launching a task is simply a matter of making an RPC to it and passing a Runnable to a thread pool, which takes in the single digits of milliseconds.
- Spark somehow runs entirely in memory while MapReduce does not. This is simply not the case. Spark's shuffle implementation works very similarly to MapReduce's: each record is serialized and written out to disk on the map side and then fetched and deserialized on the reduce side.

Q.7 Explain MLlib Data types.

❖ Local vector

- A local vector has integer-typed and 0-based indices and double-typed values, stored on a single machine.
- MLlib supports two types of local vectors: dense and sparse.
- A dense vector is backed by a double array representing its entry values, while a sparse vector is backed by two parallel arrays: indices and values. For example, a vector (1.0, 0.0, 3.0) can be represented in dense format as [1.0, 0.0, 3.0] or in sparse format as (3, [0, 2], [1.0, 3.0]), where 3 is the size of the vector.
- Java Example :

```
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
// Create a dense vector (1.0, 0.0, 3.0).
Vector dv = Vectors.dense(1.0, 0.0, 3.0);
```

```
// Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values corresponding to nonzero entries.
Vector sv = Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0});
```

❖ Labeled point

- A labeled point is a local vector, either dense or sparse, associated with a label/response.
- In MLlib, labeled points are used in supervised learning algorithms.
- We use a double to store a label, so we can use labeled points in both regression and classification. For binary classification, a label should be either 0 (negative) or 1 (positive). For multiclass classification, labels should be class indices starting from zero: 0, 1, 2, ...
- Java Example

```
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.mllib.regression.LabeledPoint;
// Create a labeled point with a positive label and a dense feature vector.
LabeledPoint pos = new LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0));
// Create a labeled point with a negative label and a sparse feature vector.
LabeledPoint neg = new LabeledPoint(1.0, Vectors.sparse(3, new int[] {0, 2}, new double[] {1.0, 3.0}));
```

❖ Local matrix

- A local matrix has integer-typed row and column indices and double-typed values, stored on a single machine. MLlib supports dense matrices, whose entry values are stored in a single double array in column major. For example, the following matrix

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}$$

is stored in a one-dimensional array [1.0, 3.0, 5.0, 2.0, 4.0, 6.0] with the matrix size (3, 2).

- Java Example

```
import org.apache.spark.mllib.linalg.Matrix;
import org.apache.spark.mllib.linalg.Matrices;
// Create a dense matrix ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
Matrix dm = Matrices.dense(3, 2, new double[] {1.0, 3.0, 5.0, 2.0, 4.0, 6.0});
```

❖ **Distributed matrix**

- A distributed matrix has long-typed row and column indices and double-typed values, stored distributively in one or more RDDs. It is very important to choose the right format to store large and distributed matrices.
- Converting a distributed matrix to a different format may require a global shuffle, which is quite expensive. Three types of distributed matrices have been implemented so far.

a) RowMatrix

» A RowMatrix is a row-oriented distributed matrix without meaningful row indices, backed by an RDD of its rows, where each row is a local vector.

» Since each row is represented by a local vector, the number of columns is limited by the integer range but it should be much smaller in practice.

» A RowMatrix can be created from a JavaRDD<Vector> instance.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.distributed.RowMatrix;
JavaRDD<Vector> rows = ... // a JavaRDD of local vectors
// Create a RowMatrix from an JavaRDD<Vector>.
RowMatrix mat = new RowMatrix(rows.rdd());
// Get its size.
long m = mat.numRows();
long n = mat.numCols();
```

b) IndexedRowMatrix

» An IndexedRowMatrix is similar to a RowMatrix but with meaningful row indices.

» It is backed by an RDD of indexed rows, so that each row is represented by its index (long-typed) and a local vector.

» An IndexedRowMatrix can be created from an JavaRDD<IndexedRow> instance, where IndexedRow is a wrapper over (long, Vector). An IndexedRowMatrix can be converted to a RowMatrix by dropping its row indices.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.distributed.IndexedRow;
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;
import org.apache.spark.mllib.linalg.distributed.RowMatrix;
JavaRDD<IndexedRow> rows = ... // a JavaRDD of indexed rows
// Create an IndexedRowMatrix from a JavaRDD<IndexedRow>.
IndexedRowMatrix mat = new IndexedRowMatrix(rows.rdd());
```

```
// Get its size.
long m = mat.numRows();
long n = mat.numCols();
// Drop its row indices.
RowMatrix rowMat = mat.toRowMatrix();
```

c) CoordinateMatrix

» A CoordinateMatrix is a distributed matrix backed by an RDD of its entries.

» Each entry is a tuple of (i: Long, j: Long, value: Double), where iis the row index, j is the column index, and value is the entry value.

» A CoordinateMatrix should be used only when both dimensions of the matrix are huge and the matrix is very sparse.

» A CoordinateMatrix can be created from a JavaRDD<MatrixEntry> instance, where MatrixEntry is a wrapper over (long, long, double). A CoordinateMatrix can be converted to an IndexedRowMatrix with sparse rows by calling toIndexedRowMatrix.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.mllib.linalg.distributed.CoordinateMatrix;
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix;
import org.apache.spark.mllib.linalg.distributed.MatrixEntry;
JavaRDD<MatrixEntry> entries = ... // a JavaRDD of matrix entries
// Create a CoordinateMatrix from a JavaRDD<MatrixEntry>.
CoordinateMatrix mat = new CoordinateMatrix(entries.rdd());
// Get its size.
long m = mat.numRows();
```

```
long n = mat.numCols();
// Convert it to an IndexRowMatrix whose rows are sparse vectors.
IndexedRowMatrix indexedRowMatrix = mat.toIndexedRowMatrix();
```

Q. 8 Explain MLlib basic statistics.

❖ Summary statistics

- We provide column summary statistics for RDD[Vector] through the function colStats available in Statistics.
- colStats() returns an instance of MultivariateStatisticalSummary, which contains the column-wise max, min, mean, variance, and number of nonzeros, as well as the total count.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.stat.MultivariateStatisticalSummary;
import org.apache.spark.mllib.stat.Statistics;
JavaSparkContext jsc = ...
JavaRDD<Vector> mat = ... // an RDD of Vectors
// Compute column summary statistics.
MultivariateStatisticalSummary summary = Statistics.colStats(mat.rdd());
System.out.println(summary.mean()); // a dense vector containing the mean value for each column
System.out.println(summary.variance()); // column-wise variance
System.out.println(summary.numNonzeros()); // number of nonzeros in each column
```

❖ Correlations

- Calculating the correlation between two series of data is a common operation in Statistics. In MLlib we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson's and Spearman's correlation.
- Statistics provides methods to calculate correlations between series. Depending on the type of input, two JavaDoubleRDDs or a JavaRDD<Vector>, the output will be a Double or the correlation Matrix respectively.

```
import org.apache.spark.api.java.JavaDoubleRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.linalg.*;
import org.apache.spark.mllib.stat.Statistics;
JavaSparkContext jsc = ...
JavaDoubleRDD seriesX = ... // a series
JavaDoubleRDD seriesY = ... // must have the same number of partitions and cardinality as seriesX
// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a method is not specified, Pearson's method will be used by default.
Double correlation = Statistics.corr(seriesX.srdd(), seriesY.srdd(), "pearson");
JavaRDD<Vector> data = ... // note that each Vector is a row and not a column
// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method. If a method is not specified, Pearson's method will be used by default.
Matrix correlMatrix = Statistics.corr(data.rdd(), "pearson");
```

❖ Stratified sampling

- Unlike the other statistics functions, which reside in MLlib, stratified sampling methods, sampleByKey and sampleByKeyExact, can be performed on RDD's of key-value pairs.
- For stratified sampling, the keys can be thought of as a label and the value as a specific attribute. For example the key can be man or woman, or document ids, and the respective values can be the list of ages of the people in the population or the list of words in the documents.
- The sampleByKey method will flip a coin to decide whether an observation will be sampled or not, therefore requires one pass over the data, and provides an expected sample size.
- sampleByKeyExact requires significant more resources than the per-stratum simple random sampling used in sampleByKey, but will provide the exact sampling size with 99.99% confidence. sampleByKeyExact is currently not supported in python.
- sampleByKeyExact() allows users to sample exactly $\lceil f_k \cdot n_k \rceil \forall k \in K$ items, where f_k is the desired fraction for key k, n_k is the number of key-value pairs for key k, and K is the set of keys. Sampling without replacement requires one additional pass over the RDD to guarantee sample size, whereas sampling with replacement requires two additional passes.

```
import java.util.Map;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
JavaSparkContext jsc = ...
JavaPairRDD<K, V> data = ... // an RDD of any key value pairs
Map<K, Object> fractions = ... // specify the exact fraction desired from each key
```

```
// Get an exact sample from each stratum
JavaPairRDD<K, V> approxSample = data.sampleByKey(false, fractions);
JavaPairRDD<K, V> exactSample = data.sampleByKeyExact(false, fractions);
```

❖ Hypothesis testing

- Hypothesis testing is a powerful tool in statistics to determine whether a result is statistically significant, whether this result occurred by chance or not.
- MLlib currently supports Pearson's chi-squared (χ^2) tests for goodness of fit and independence.
- The input data types determine whether the goodness of fit or the independence test is conducted. The goodness of fit test requires an input type of Vector, whereas the independence test requires a Matrix as input.
- MLlib also supports the input type RDD[LabeledPoint] to enable feature selection via chi-squared independence tests.
- Statistics provides methods to run Pearson's chi-squared tests. The following example demonstrates how to run and interpret hypothesis tests.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.linalg.*;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.stat.Statistics;
import org.apache.spark.mllib.stat.test.ChiSqTestResult;

JavaSparkContext jsc = ...
Vector vec = ... // a vector composed of the frequencies of events
```

```
// compute the goodness of fit. If a second vector to test against is not supplied as a parameter,
// the test runs against a uniform distribution.
ChiSqTestResult goodnessOfFitTestResult = Statistics.chiSqTest(vec);
// summary of the test including the p-value, degrees of freedom, test statistic, the method used,
// and the null hypothesis.
System.out.println(goodnessOfFitTestResult);
Matrix mat = ... // a contingency matrix
// conduct Pearson's independence test on the input contingency matrix
ChiSqTestResult independenceTestResult = Statistics.chiSqTest(mat);
// summary of the test including the p-value, degrees of freedom...
System.out.println(independenceTestResult);
JavaRDD<LabeledPoint> obs = ... // an RDD of labeled points
// The contingency table is constructed from the raw (feature, label) pairs and used to conduct
// the independence test. Returns an array containing the ChiSquaredTestResult for every feature
// against the label.
ChiSqTestResult[] featureTestResults = Statistics.chiSqTest(obs.rdd());
int i = 1;
for (ChiSqTestResult result : featureTestResults) {
  System.out.println("Column " + i + ":");
  System.out.println(result); // summary of the test
  i++;
}
```

❖ **Random data generation**

- Random data generation is useful for randomized algorithms, prototyping, and performance testing. MLlib supports generating random RDDs with i.i.d. values drawn from a given distribution: uniform, standard normal, or Poisson.
- RandomRDDs provides factory methods to generate random double RDDs or vector RDDs. The following example generates a random double RDD, whose values follows the standard normal distribution N(0, 1), and then map it to N(1, 4).

```
import org.apache.spark.SparkContext;
import org.apache.spark.api.JavaDoubleRDD;
import static org.apache.spark.mllib.random.RandomRDDs.*;

JavaSparkContext jsc = ...
// Generate a random double RDD that contains 1 million i.i.d. values drawn from the
// standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
JavaDoubleRDD u = normalJavaRDD(jsc, 1000000L, 10);
// Apply a transform to get a random double RDD following `N(1, 4)`.
JavaDoubleRDD v = u.map(
 new Function<Double, Double>() {
  public Double call(Double x) {
   return 1.0 + 2.0 * x;
  }
 });
```